Data-Flow/Dependence Profiling for Structured Transformations

Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, Fabrice Rastello

Data-Flow/Dependence Profiling for Structured Transformations

Fabian Gruber
Manuel Selva
Diogo Sampaio
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

Christophe Guillon
Antoine Moynault
STMicroelectronics

Louis-Noël Pouchet
Colorado State University

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

Abstract

Profiling feedback is an important technique used by developers for performance debugging, where it is usually used to pinpoint performance bottlenecks and also to find optimization opportunities. Assessing the validity and potential benefit of a program transformation requires accurate knowledge of the data flow and dependencies, which can be uncovered by profiling a particular execution of the program.

In this work we develop POLY-PROF, an end-to-end infrastructure for dynamic binary analysis, which produces feedback about the potential to apply complex program rescheduling. Our tool can handle both inter- and intraprocedural aspects of the program in a unified way, thus providing interprocedural transformation feedback.

Keywords Performance Feedback, Polyhedral Model, Loop Transformations, Compiler Optimization, Binary, Instrumentation, Dynamic Dependence Graph

1 Introduction

The most effective program transformations for improving performance or energy consumption are typically based on rescheduling instructions so as to expose data locality and/or parallelism. Optimizing compilers typically attempt, via static analysis, to build a representation of the program precise enough to enable useful program transformations. A key issue faced when analyzing general-purpose languages is the ambiguities introduced by the language itself: for example the use of pointers typically restricts the ability to precisely characterize the data being accessed, in turn triggering conservative and approximate dependence analysis [27, 40, 68]. Such frameworks therefore rely on conservative assumptions, limiting the ability to reason on the legality of complex program restructuring. In this work we specifically target *structured transformations*, that is, a (possibly interprocedural) reordering of operations involving complex sequences of multidimensional loop transformations, including (partial) loop fusion/fission, loop tiling [11], etc.

When a region of the source program fits specific syntactic and semantics restrictions, such as avoiding function calls, using only arrays as data structures, and very simple conditional statements with no indirections [16, 20], transformation frameworks such as the polyhedral model [11] showed that multidimensional loop nest rescheduling can be successfully implemented, leading to significantly improved performance [28, 53, 68]. The input program can be massaged manually via function inlining, loop rewriting/normalization, etc. to enable such static analyses to succeed, but this is a rare scenario: In full programs and in particular those relying on external binaries visible only to the linker in compiled form, often data allocation and even the full call stack is inaccessible to static analysis.

Dynamic analysis frameworks [9, 21, 47, 63] address this limitation by reasoning on a particular *execution* of the program, through the analysis of its execution trace. That is, disambiguation of dependence information and accessed data is addressed by monitoring the stream of operations and accessed addresses. In general, the result is only valid for that particular execution. Such systems provide some feedback to the user, pinpointing the probable absence of dependencies along some loop, thus highlighting potential parallelism [37, 67, 70, 74] or SIMD vectorization [33].

The contributions of this work are: 1. The development of an inter-procedural intermediate representation that compacts program traces effectively while being amenable to polyhedral optimization, and all associated algorithms and implementation to make the process scalable to full programs. 2. POLY-PROF, a tool that provides optimization feedback using binary translation and instrumentation. POLY-PROF is the first framework that provides feedback on structured transformation potential for arbitrary binaries.

2 Overview of POLY-PROF

In contrast to, for instance, basic loop parallelization, structured transformations can be found even in the presence of data dependencies between loop iterations, dramatically broadening the class of loop transformations that can be applied on the program. This, however, requires detailed information about such dependencies. That is, instead of just proving the absence of dependencies one must capture and represent them in order to be able to find patterns and reason about their structure.

An overview of POLY-PROF is shown in Fig. 1. Taking a carefully generated execution trace of the program, the first objective of POLY-PROF is to construct a *useful and analyzable* representation of the trace. Building such a representation constitutes our first key set of contributions, and covers the first three stages below. We then extended a polyhedral compiler. The goal here is not to provide automated

transformations, but instead to assist the user in figuring out where optimizations should be implemented, and importantly which ones. This corresponds to the fourth stage below that forms the second contribution of this work.

Interprocedural control structure To eventually reconstruct a compact and useful representation of the program trace, the first stage starts by dynamically computing the intraprocedural control flow graphs (CFGs) and the interprocedural call graph (CG) from an optimized binary. This information is then used to construct the *interprocedural loop context tree* which combines the notions of loop-nesting-trees and their equivalent for call graphs, the *recursive-component-set*, both of which are input to the next stage. Their structure and construction are detailed in section 3.

Dynamic Dependence Graph This stage generates the actual dynamic dependence graph (DDG) [50], a trace representing both the dynamic execution of a program's instructions as well as their data dependencies. It uses the interprocedural loop context tree to construct *dynamic interprocedural iteration vectors*, a new interprocedural representation of the program execution which unifies two abstractions: the intraprocedural schedule tree [35, 72] and the interprocedural calling-context-tree [2]. Both the DDG and the dynamic interprocedural iteration vector are described in section 4.

Compact polyhedral DDG The third stage compacts, or folds, the DDG into a polyhedral program representation where, essentially, sequences of individual operations in the trace are merged together into sets of points. Intuitively, when a loop is executed it generates one execution of a statement for each loop iteration and therefore one point in the trace for each execution. This stage amounts to folding those points back into a loop-equivalent representation, which itself can be the input of a polyhedral compiler. However we proceed in a general case, where folding occurs across multi-level loops but also, possibly recursive, procedure calls, leading to interprocedural folding. Details are presented in [29], and an outline of the process is presented in Sec. 5.

DDG polyhedral feedback The last stage of POLY-PROF consists of a customized state-of-the-art polyhedral toolchain, for the purpose of analyzing very large polyhedral programs. It analyzes the polyhedral DDG and suggests sequences of program transformations needed to expose multi-level parallelism and/or improved data locality, along with numerous statistics on the original and potentially transformed program including SIMDization potential. Section 6 describes this last step as well as how POLY-PROF provides human interpretable feedback to the user.

Section 7 illustrates this feedback through a few case studies. Section 8 thoroughly evaluates POLY-PROF on the Rodinia benchmarking suite. Related work is discussed in Section 9 before concluding.

3 Interprocedural Control Structure

As most of the execution time of a program is usually spent in loops, POLY-PROF has been designed to find loop-based transformations. In practice, however, as illustrated in Ex. 1

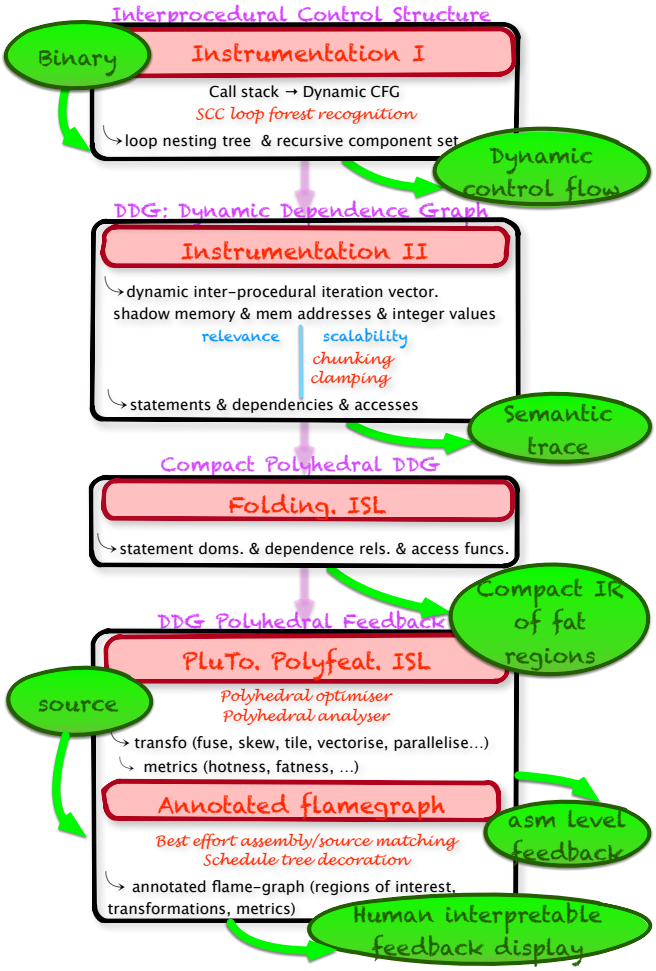


Figure 1. Overview over the POLY-PROF framework.

of Fig. 3, interesting loop nests are often spread across multiple functions over long call chains, obfuscating them to traditional static analysis. POLY-PROF is designed to be able to represent both classical loops as well as function calls and returns in a unified way with polyhedra [23, 75].

POLY-PROF has been designed to handle several scenarios. 1. Nested loops containing function calls with side effects: here calls can be viewed as atomic instructions, but profiling the storage locations they access or modify and compute the dependencies between each call and the rest of the loop body is required. 2. Loops containing function calls that themselves contain loops, as in the Ex. 1 of Fig. 3: here the objective is to view the whole interprocedural region as a multidimensional loop nest. 3. Recursive functions, as in Ex. 2 of Fig. 3: here the primary objective is to avoid the depth of the generated representation to grow proportionally with the depth of the call stack; A secondary objective is to detect any underlying regular loop structure amenable to polyhedral compilation (after recursion-to-iteration conversion).

This section explains how to go from the execution of a program to a representation of its interprocedural control structure.

The first step of POLY-PROF is to extract the inter- and intraprocedural static control structure from a program in compiled binary form. This static control structure of the program will then be used in a later stage of POLY-PROF to transform the stream of raw events (`jump`/`call`/`return`) gathered during execution into loop events (`entry`/`iterate`/`exit`).

POLY-PROF performs this through instrumentation of `jump`, `call`, `return` and a few other instructions to dynamically trace any transfer of control during execution. It thus "dynamically" extracts the control-flow-graph (CFG) of each function and then proceeds to statically build the *loop-nesting-tree*. It also dynamically extracts the call-graph (CG) of the whole program and builds the counterpart of loop-nesting-trees for call-graphs, the *recursive-component-set*. An advantage of our approach is that only the part of a program that is actually executed will be analyzed. For large programs with only a small performance-critical part this can substantially reduce the amount of work for the analyzer.

3.1 Control-flow-graph and loop-nesting-tree

For the profiled CFG, the loop detection algorithm used by POLY-PROF is the one described by Havlak [31]. As formalized by Ramalingam in [58], a loop-nesting-forest can be recursively defined (the actual algorithm is almost linear) as follows: 1. Each strongly connected component (SCC) of the CFG containing at least one cycle constitutes the *region* of an outermost loop; 2. for each loop, one of its *entry* nodes is designated the *header* node of the loop; 3. all edges within the loop that target its header node are called *back-edges*; 4. "removing" all back-edges from the CFG allows one to recursively define the next level of sub-loops.

An example CFG and its corresponding loop-nesting-tree is given in Figures 2a & 2b. Here, the CFG can be partitioned into one SCC (giving rise to loop L_1 with B as its header) plus two separated basic-blocks A and E. The region of L_1, once its back-edge (D, B) has been removed, contains one sub-SCC formed by the loop L_2 and the basic-block B. Among the two entry nodes C and D of L_2 only one, C, is selected to be its header. As depicted in Alg. 1, those notions (region, header, back-edge) are important as they allow associating loop events (E: entry, I: iterate, and X: exit) with the stream of raw control events (`jump`, `call`, `return`) that are produced by our instrumented execution. Here, generation of loop events is driven by the control event "`jump`" that is emitted by our instrumentation. Whenever we detect the start of a new iteration of a given loop (Line 8), all live sub-loops are considered exited, that is, we emit an "`exit`" event (Line 4). This is especially important for recursive loops as further explained in Alg. 2.

3.2 Call-graph and recursive-component-set

To uniquely identify each dynamic execution of an instruction, also called a *dynamic instruction*, POLY-PROF uses interprocedural iteration vectors. This is described in more detail in Section 4. Note, that the modeling of programs containing recursive function calls with calling-context-paths is

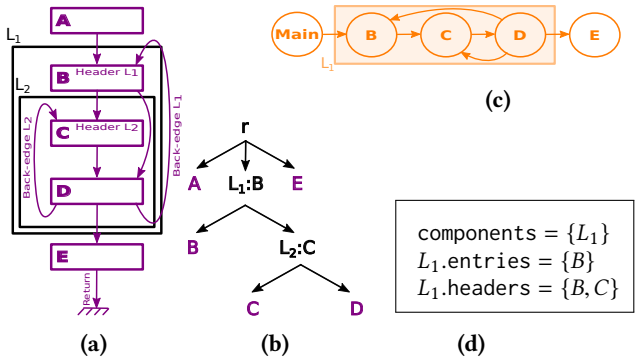


Figure 2. (a/b) Example CFG and associated loop-nesting-tree; (c/d) Example CG and associated recursive-components-set.

Algorithm 1 CFG-loop events generated from a `jump` event.

Input:
- `event`: Branch event.
- `inLoops`: Stack of currently live loops. From outermost to innermost.

Emitted events:
- `E(L, H)`: entry into loop L; H is the header.
- `I(L, H)`: iteration of loop L; H is the header.
- `X(L, B)`: exit of loop L, jumping to basic-block B
- `N(B)`: local jump to basic-block B.

```
1  if event.type is local-jump:
2    B:=event.dstBB
3    while L:=inLoops.peek() and L.isCFG and B not in L:
4      L.visiting:=false; inLoops.pop(); emit X(L,B)
5    if B is loop-header:
6      H:=B; L:=loop(H)
7      if L.visiting=false:
8        L.visiting:=true; inLoops.push(L); emit E(L,H)
9      else: emit I(L,H)
10   emit N(B)
11 else:
12   ... # Alg. 2
```

memory inefficient since the length of the paths is proportional to the depth of the recursion. While recursive function calls are found to be very uncommon in performance critical code [1], we do need to handle them to ensure robustness. POLY-PROF handles recursiveness in an elegant way that we believe to be useful beyond the restricted scope of this paper, for example to detect properties of tree-recursive calls. The main data structure for treating recursive control flow is the *recursive-component-set* which is for the call-graph what the loop-nesting-tree is for the control-flow-graph.

Before providing more details, we go through the illustrating Ex. 2 of Fig. 3. Here, the edge from B_3 to B_0 (in orange) is not a CFG-edge but a recursive call to B from call site B_3. The recursive-component-set computed from the CG contains one SCC with a cycle made up of a single function B. This example raises several questions: 1. *Should C_0 be part of the associated recursive loop?* It actually depends on the context: It should, when called from B_1, while it should not when called from D_0. 2. *What about B_5?* One should observe that B_5 will be executed as many times as the number of recursive calls to B: In other words, B_5 should be part of a loop.

[1] the Rodinia benchmark suite, for example, does not use any recursion

To conclude, while CG-cycles clearly represent potential dynamic loop structures, a CG-edge does not have the same semantic as a CFG-edge. In particular, *a* `call` *will never exit a recursive loop*. For POLY-PROF, the recursive loop "L_1" of Ex. 2 is a dynamic notion defined as follows: 1. Entering L_1 is characterized by a call to B (step 1). 2. L_1 is exited when this (first) call gets unstacked, that is, when the number of returns reaches the number of calls (step 22). 3. Everything executed in between is part of the loop and iteration, and the corresponding increment of induction variables takes place whenever a call/return to/from B occurs (steps 10,15,20,and 21). As one can see, once the recursive-component-set has been computed from the CG, the only relevant information dynamically used to devise recursive-loop events corresponds to the *header* functions, here B.

As already stated, the recursive-component-set is for the CG what the loop-nesting-tree is for the CFG. In a similar way it can be recursively defined as follows:

1. Find all the top-level SCCs with at least one cycle in the CG. Each gives rise to a distinct recursive-*component*.
2. For each component, determine all its *entry* nodes.
3. Repeat the following phases until no more cycles exist:
   a. For a given SCC, choose an entry node and add it to the *headers*-set of the recursive-component (top-level SCC) it belongs to.
   b. Remove all edges inside this SCC that point to this node.

Havlak's loop-nesting-forest construction algorithm can easily be adapted to build the recursive-component-set in almost linear time. The end result of the algorithm is a possibly empty set of recursive-components where each recursive-component has a non-empty set of headers plus a non-empty set of entries associated with it. Alg. 2 uses this data structure to associate loop events (entry, iterate, and exit) to control events (`call` and `return`). Here: 1. entering a recursive loop is characterized by a call to a component's entry function (Line 4); 2. a new iteration is started whenever a call/return to/from one of the components' header occurs (Line 6); 3. an exit occurs when all the iterating calls to the headers have been unstacked, that is, when the number of returns equals the number of calls, and we are returning from the original function that entered the loop (Line 18). Tracking the state of the call stack is done with the following two data structures: L.`stackcount` represents for a recursive-component L a counter of the number of calls-to a header minus the number of returns from it; L.`entry` represents the function through which L was entered.

4 DDG: Dynamic dependence graph

The objective of the second stage of POLY-PROF ("Instrumentation II") is to profile the dynamic dependence graph of a given execution, that is, to build a graph that has one vertex for every dynamic instruction and one edge for every data dependence. Because we want to enable feedback with structured loop transformations, we need to map this graph to a "geometric" space that reflects the structural properties of the program. To this end, we tag each dynamic instruction with its *iteration vector* (IV). The IVs uniquely identify each dynamic instruction and naturally span a geometric space. A data dependency is then simply represented as the pair of the IVs of the producer and the consumer.

Algorithm 2 Different recursive-loop events generated from a `call` or a `return` event.

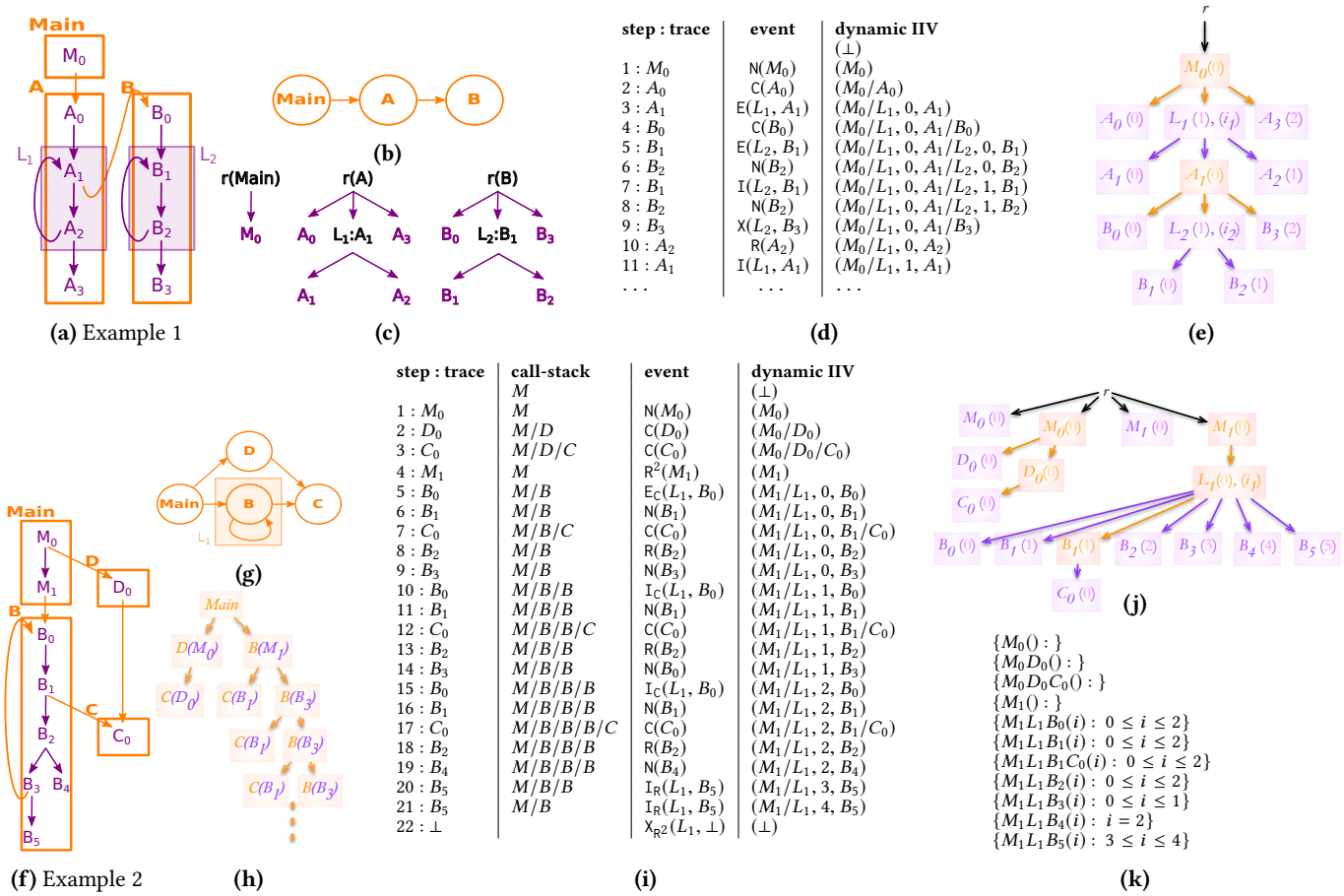
Input:
- `event`, `inLoops`: same as for Alg. 1

Emitted events:
- E_C(**L, B**): `call`-to a function header of recursive-component L and entry into the corresponding loop. B is the current basic-block after the call.
- I_C(**L, B**) / I_R(**L, B**): `call`-to / `return`-from a function header of recursive-component L and iteration of the corresponding loop. B is the current basic-block after the call/return.
- X_R(**L, B**): `return` from a function header of recursive-component L and exit from that loop.

```
1 if event.type is call:
2   F := event.callee; B := event.dstBB
3   L := F.recursive_component
4   if F is recursive-component entry and L.entry==undef:
5     L.entry := F; inLoops.push(L); emit E_C(L,B)
6   elif F is recursive-component header:
7     while L':=inLoops.peek() and L' in L:
8       L'.visiting := false; inLoops.pop();
9       emit X(L',B)
10    L.stackcount++; emit I_C(L,B)
11  else: emit C(F,B)
12 if event.type is return:
13  F := event.callee; B := event.dstBB
14  while L':=inLoops.peek() and L'.isCFG and L' in F
15    L'.visiting := false; inLoops.pop(); emit X(L',B)
16  L := F.recursive_component
17  if F is recursive-component entry and
18     L.stackcount == 0 and L.entry == F:
19      L.entry := undef; emit X_R(L,B)
20  elif F is recursive-component header:
21    L :=  F.recursive_component; L.stackcount--;
22    emit I_R(L,B)
23 else:
24  ... # Alg. 1
```

To handle interprocedural programs we also need a notion of calling context that is scalable in the presence of recursive calls. Our *dynamic interprocedural iteration vector* (dynamic IIV) described in this section addresses those objectives by unifying two notions: 1. Kelly's mapping which describes intraprocedural IVs and is used by the polyhedral framework [35]; 2. calling-context-paths used by profiling feedback tools. We first briefly recall those two notions.

Kelly's mapping For a given function, Kelly's mapping can be explained using a form of *schedule tree* [72] as shown in Fig. 4. Here a schedule tree is nothing else than a decorated loop-nesting-forest. The first decoration consists of associating a "static" index to each loop and basic-block: Recall the recursive characterization of loops given by Ramalingam in the previous section. For a given loop region (e.g. L_j in the schedule-tree of the fused version in Fig. 4), its sub-regions (once back-edges have been removed – here statements S and T) form a directed-acyclic-graph (reduced DAG represented in dashed in Fig. 4) that can be numbered using a topological order. This numbering is used to index the corresponding nodes at every level of the loop-nesting-tree (e.g. $S(0)$ and $T(1)$ for the fused schedule or $L_i(0)$ and $L_{i'}(1)$ for the fissioned one). The second decoration consists of associating a canonical induction variable, that is, an induction variable that starts at value 0 and increments by 1, to each loop-vertex. As an example the loop-vertex associated with L_j is labeled



step : trace	event	dynamic IIV
		$(\perp)$
$1 : M_0$	$\mathtt{N}(M_0)$	(M_0)
$2 : A_0$	$\mathtt{C}(A_0)$	(M_0/A_0)
$3 : A_1$	$\mathtt{E}(L_1, A_1)$	$(M_0/L_1, 0, A_1)$
$4 : B_0$	$\mathtt{C}(B_0)$	$(M_0/L_1, 0, A_1/B_0)$
$5 : B_1$	$\mathtt{E}(L_2, B_1)$	$(M_0/L_1, 0, A_1/L_2, 0, B_1)$
$6 : B_2$	$\mathtt{N}(B_2)$	$(M_0/L_1, 0, A_1/L_2, 0, B_2)$
$7 : B_1$	$\mathtt{I}(L_2, B_1)$	$(M_0/L_1, 0, A_1/L_2, 1, B_1)$
$8 : B_2$	$\mathtt{N}(B_2)$	$(M_0/L_1, 0, A_1/L_2, 1, B_2)$
$9 : B_3$	$\mathtt{X}(L_2, B_3)$	$(M_0/L_1, 0, A_1/B_3)$
$10 : A_2$	$\mathtt{R}(A_2)$	$(M_0/L_1, 0, A_2)$
$11 : A_1$	$\mathtt{I}(L_1, A_1)$	$(M_0/L_1, 1, A_1)$
...	...	...

step : trace	call-stack	event	dynamic IIV
	M		$(\perp)$
$1 : M_0$	M	$\mathtt{N}(M_0)$	(M_0)
$2 : D_0$	M/D	$\mathtt{C}(D_0)$	(M_0/D_0)
$3 : C_0$	$M/D/C$	$\mathtt{C}(C_0)$	$(M_0/D_0/C_0)$
$4 : M_1$	M	$\mathtt{R}^2(M_1)$	(M_1)
$5 : B_0$	M/B	$\mathtt{E}_\mathtt{C}(L_1, B_0)$	$(M_1/L_1, 0, B_0)$
$6 : B_1$	M/B	$\mathtt{N}(B_1)$	$(M_1/L_1, 0, B_1)$
$7 : C_0$	$M/B/C$	$\mathtt{C}(C_0)$	$(M_1/L_1, 0, B_1/C_0)$
$8 : B_2$	M/B	$\mathtt{R}(B_2)$	$(M_1/L_1, 0, B_2)$
$9 : B_3$	M/B	$\mathtt{N}(B_3)$	$(M_1/L_1, 0, B_3)$
$10 : B_0$	$M/B/B$	$\mathtt{I}_\mathtt{C}(L_1, B_0)$	$(M_1/L_1, 1, B_0)$
$11 : B_1$	$M/B/B$	$\mathtt{N}(B_1)$	$(M_1/L_1, 1, B_1)$
$12 : C_0$	$M/B/B/C$	$\mathtt{C}(C_0)$	$(M_1/L_1, 1, B_1/C_0)$
$13 : B_2$	$M/B/B$	$\mathtt{R}(B_2)$	$(M_1/L_1, 1, B_2)$
$14 : B_3$	$M/B/B$	$\mathtt{N}(B_0)$	$(M_1/L_1, 1, B_3)$
$15 : B_0$	$M/B/B/B$	$\mathtt{I}_\mathtt{C}(L_1, B_0)$	$(M_1/L_1, 2, B_0)$
$16 : B_1$	$M/B/B/B$	$\mathtt{N}(B_1)$	$(M_1/L_1, 2, B_1)$
$17 : C_0$	$M/B/B/B/C$	$\mathtt{C}(C_0)$	$(M_1/L_1, 2, B_1/C_0)$
$18 : B_2$	$M/B/B/B$	$\mathtt{R}(B_2)$	$(M_1/L_1, 2, B_2)$
$19 : B_4$	$M/B/B/B$	$\mathtt{N}(B_4)$	$(M_1/L_1, 2, B_4)$
$20 : B_5$	$M/B/B$	$\mathtt{I}_\mathtt{R}(L_1, B_5)$	$(M_1/L_1, 3, B_5)$
$21 : B_5$	M/B	$\mathtt{I}_\mathtt{R}(L_1, B_5)$	$(M_1/L_1, 4, B_5)$
$22 : \perp$		$\mathtt{X}_{\mathtt{R}^2}(L_1, \perp)$	$(\perp)$

$\{M_0() : \}$
$\{M_0 D_0() : \}$
$\{M_0 D_0 C_0() : \}$
$\{M_1() : \}$
$\{M_1 L_1 B_0(i) : \ 0 \le i \le 2\}$
$\{M_1 L_1 B_1(i) : \ 0 \le i \le 2\}$
$\{M_1 L_1 B_1 C_0(i) : \ 0 \le i \le 2\}$
$\{M_1 L_1 B_2(i) : \ 0 \le i \le 2\}$
$\{M_1 L_1 B_3(i) : \ 0 \le i \le 1\}$
$\{M_1 L_1 B_4(i) : \ i = 2\}$
$\{M_1 L_1 B_5(i) : \ 3 \le i \le 4\}$

Figure 3. (a/f) Code skeletons; Purple is used to represent CFG, while orange is used to represent CG; (b/g) Call-graph and recursive-component-set; (c) Loop-nesting-forests; (h) Calling-context-tree with call-sites information; (d/i) Example trace, associated loop events, and dynamic interprocedural interation vectors. $\mathtt{R}^2$ stands for two consecutive returns; (e/j) Dynamic schedule tree; (k) Folded domains.

with L_j as well as the static index 0 followed by its canonical variables j, resulting in $L_j(0), (j)$. For any given statement, an IV with Kelly's mapping is nothing else than the vector given by the path from the root to its corresponding leaf, called its *iteration vector*. Fig. 4 shows this mapping both in its textual form using region names and numerical form using indices. As one can see, an interesting property of the numerical form of this mapping is that the scheduling of the original code is given by the lexicographical order of the so obtained iteration vectors (unique per dynamic instance).

Calling-context-tree Differently from the schedule tree, the calling-context-tree [2] (CCT) is only enumerative (it does not contain any loop indices) and reflects a dynamic behavior. In other words, it encodes the dynamic-call-tree in a "compact" way. The calling-context-tree of the example in Fig. 3f is reported in Fig. 3h. This figure is slightly different from the "original" CCT, but corresponds to the current practice: to differentiate two calls to a common function from different basic-blocks, callees are labeled with call sites (in purple and under parenthesis in Fig. 3h). In this example, a calling-context-path (for example $M_1/B_3 \ldots B_3/B_1/C$) can be as long as the number of recursive calls to B, but the calling context is fully encoded making it possible to differentiate the different contexts within which basic-block C_0 is executed. Obviously, looking at this example, one wants to fold all the repeated calls from B_1 to C.

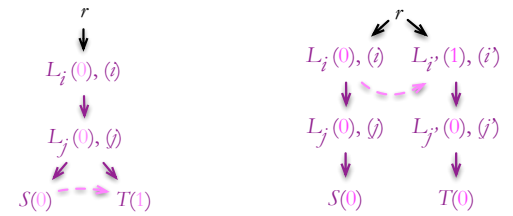
Dynamic IIV The dynamic IIV is basically a combination of Kelly's mapping and the CCT. Similarly to Kelly's mapping, the dynamic IIV alternates between context-ids and canonical induction variables. Differently from Kelly's mapping, but more like the CCT, each context-id in a dynamic IIV is a, possibly empty, stack of calling contexts and the identifier for a basic block.

Examples of IIVs are shown in Fig. 3a: Function A contains a loop L_1 that contains a call to function B, itself containing a loop L_2. In this interprocedural example one clearly wants to see that basic-block B_1 in loop L_2 belongs to a two-dimensional nested loop. This is reflected by our dynamic IIV (see steps 5,7 of Fig. 3d) which will be $(M_0/L_1, i_1, A_1/L_2, i_2, B_1)$ where i_1 (resp. i_2) are the canonical induction variables of loop L_1 (resp. L_2). Here, our context-ids are loop-ids (e.g. L_2) or statement-ids (e.g. B_0) and each context-id is decorated with a, possibly empty, call stack (e.g. A_1).

```
1 for (i=0;i<n;i++)
2   for (j=0;j<=i;j++)
3     { S; T }
```

```
1 for (i=0;i<n;i++)
2   for (j=0;j<=i;j++)
3     { S }
4 for (i'=0;i'<n;i'++)
5   for (j'=0;j'<=i';j'++)
6     { T }
```

(a) *Nested loop before and after fission*



(b) *Corresponding schedule trees (reduced DAG in dashed)*

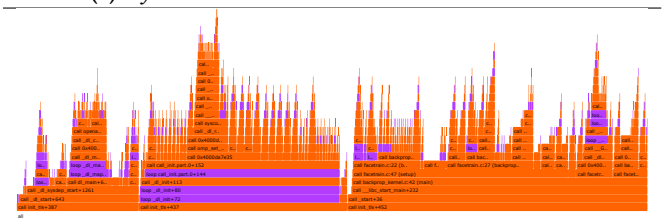
S	$\rightarrow$	$[L_i, i, L_j, j, S]$	S	$\rightarrow$	$[L_i, i, L_j, j, S]$
	$\rightarrow$	$[0, i, 0, j, 0]$		$\rightarrow$	$[0, i, 0, j, 0]$
T	$\rightarrow$	$[L_i, i, L_j, j, T]$	T	$\rightarrow$	$[L_{i'}, i', L_{j'}, j', T]$
	$\rightarrow$	$[0, i, 0, j, 1]$		$\rightarrow$	$[1, i', 0, j', 0]$

(c) *Corresponding Kelly's mapping / iteration vector (IV)*

Figure 4. Schedule tree (pink: static index; purple: induction vars) and Kelly's mapping

	interprocedural	loops
sched. tree / IV	✗	✓
CCT / calling-context-path	✓	✗
dyn. sched. tree / dynamic IIV	✓	✓

(a) *Dynamic* schedule tree ≡ schedule tree ∪ CCT



(b) *Annotated flame-graph (*`backprop` *with libc)*

Figure 5. *Dynamic* schedule tree

Similar to Kelly's mapping one can also construct a dynamic schedule tree from the dynamic IIVs of a program execution. Also, note that the schedule tree is for the dynamic IIVs what the calling-context-tree is for the set of calling context paths. The schedule tree for our example is shown in Fig. 3e. The relationship between these three structures is summarized in Fig. 5. As described in more details further, POLY-PROF exposes the schedule tree to the programmer in the form of a flame-graph [26] where the root of the tree is on the bottom and leaves on the top.

Ex. 3f illustrates how the recursive-component-set is used to "fold" calling-context-paths in the presence of recursive calls. Here, M_1 is the caller site (step 4) before entering the recursive loop L_1 (step 5). Looking at C_0, its multiple instances are indexed by the corresponding recursive-loop induction variable i_1 (respectively 0,1, and 2 at steps 7, 12, and 17). Within the loop it gets executed when C is called from B_1. The associated dynamic IIV is $(M_1/L_1, i_1, B_1/C_0)$. As already mentioned, B_5 is also part of the loop: Indeed there are as many instances of B_5 as there are calls to B from B_3 (2 calls at steps 10,15). Observe that the value of i_1 does not reflect the size of the call stack: It does not go up and down. It keeps increasing. The main reason for doing so is related to our polyhedral back-end: 1. we want our indexing to be lexicographically increasing; 2. we want to match to the polyhedral formalism where iterators can be represented using simple canonical induction variables. To do so, any `return` (except the last that exits the recursive loop) associated with a `call` to B (steps 20,21) leads to incrementing i_1.

As previously described, loop events allow updating the dynamic IIV. This is done as detailed in Alg. 3. Here, `diiv` corresponds to the dynamic IIV, where the rightmost is the innermost dimension. Each dimension contains two fields, the induction variable `IV`, followed by the context variable `CTX`: As an example, $\texttt{I}(L_1, A_1)$ applied to $(M_0/L_1, 0, A_2)$ increments the `IV` of L_1 and updates the `CTX` to A_1, leading to $(M_0/L_1, 1, A_1)$. `CTX.last` represents the last element of the context variable: As an example, if `CTX` $= M_0/D_0/C_0$, `CTX.last` $= C_0$. `CTX.push`(B) appends B to the context variable: As an example $\texttt{C}(C_0)$ applied to $(M_1/L_1, 0, B_1)$ leads to $(M_1/L_1, 0, B_1/C_0)$. `CTX.pop()` does the reverse: As an example $\texttt{R}(M_1)$ applied to (M_0/D_0) pops D_0 and updates the last element to M_1 leading to a `CTX` of (M_1). `addDimension`$(0, B)$ adds those two fields to the vector: As an example $\texttt{E}_\texttt{C}(L_1, B_0)$ applied to (M_1), appends L_1 to `CTX` and adds a dimension with innermost context set to B_0, leading to $(M_1/L_1, 0, B_0)$. `removeDimension()` does the reverse: For example, $\texttt{X}(L_2, B_3)$ applied to $(M_0/L_1, 0, A_1/L_2, 1, B_2)$ leads to $(M_0/L_1, 0, A_1/B_3)$.

Algorithm 3 Updating of dynamic IIV (`diiv`)

Input:
- `event`: Branch event.
- `diiv`: Current dynamic IIV.

Output:
- Updated dynamic IIV

```
1 if event == C(B) or E_C(L,B):
2   diiv.innermost.CTX.push(L)
3 if event == E(L,B) or E_C(L,B):
4   diiv.innermost.CTX.last := L
5   diiv.addDimension(0,B)
6 if event == X(L,B) or X_R(L,B):
7   diiv.removeDimension()
8   diiv.innermost.CTX.last:=B
9 if event == I(L,B) or I_C(L,B) or I_R(L,B):
10  diiv.innermost.IV++
11  diiv.innermost.CTX.last := B
12 if event == R(B):
13  diiv.innermost.CTX.pop()
14  diiv.innermost.CTX.last := B
```

5 Compact Polyhedral DDG

In the DDG, every dynamic instruction is represented by one vertex and every data dependence by one edge. A program running on a modern CPU for just a few seconds can create billions of nodes in the DDG. Our *folding algorithm* compresses those large graphs into a form that is amenable to polyhedral analysis performed by our POLY-PROFS back-end. The details are presented in a separate paper [29]. Only the

```
1 for (j = 1; j <= n2)
2     sum = 0.0;
3     for (k = 0; k <= n1)
4         tmp1 = load(&conn + k)}     // I1
5         tmp2 = load(tmp1 + j)       // I2
6         tmp3 = load(&l1 + k)        // I3
7         sum = sum + tmp2 * tmp3     // I4
8         k = k + 1                   // I5
9     tmp4 = call squash(sum);        // I6
10    store(&l2 + j,  tmp4)           // I7
11    j = j + 1                       // I8
```

Figure 6. Pseudo-assembler for first kernel of backprop benchmark

I1 → I2		I2 → I4		I4 → I4	
IV (cj,ck)	Label (cj',ck')	*IV* (cj,ck)	Label (cj',ck')	*IV* (cj,ck)	Label (cj',ck')
(0,0)	(0,0)	(0,0)	(0,0)		
(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,0)
…	…	…	…	…	…

Table 1. Dependency input stream from example in Fig. 6

main properties are outlined here. Our folding algorithm borrows ideas from trace compression [36] with the notable difference that is based on a geometric approach and is able to perform over-approximations as briefly described below.

Folding interface The output of the second stage of POLY-PROF (Instrumentation II) that feeds the folding stage is a stream of IIVs and labels. The IIV representation used at this stage slightly differs from the previous section. IIVs are split into two parts: 1. The *context* corresponds to the non-numerical part of the vector; 2. The *coordinates* correspond to the numerical part. Folding is then performed for each context separately. The folding algorithm takes as input for each dynamic instruction and each dynamic dependence: 1. $\vec{I}$: its iteration vector, which uniquely identifies it (i.e., its unique coordinates). 2. *Label*: a vector $a(\vec{I})$ of associated integer values. And as an output it produces: 1. a union of polyhedra that represent the set of all $\vec{I}$. 2. for each polyhedron P, an affine function $\mathcal{A}$ such that for all $\vec{I} \in P$, $\mathcal{A}(\vec{I}) = a(\vec{I})$. For dynamic instructions, $a(\vec{I})$ are the integer and pointer values produced by the instruction, if any. For dependencies, $a(\vec{I})$ are the IV of the producer instruction.

Dependency recognition Some of the streams compressed by the folding stage are dependencies. To illustrate its functionality on dependencies, consider `bpnn_layerforward`, a kernel of the `backprop` benchmark from the Rodinia benchmark suite. Part of the input stream and corresponding folded output dependencies can be seen in Tables 1 and 2. Note that iterators used by POLY-PROF are canonical ones (here cj and ck) computed on the fly through program instrumentation, that do not necessarily match the ones present in the original code (e.g. `j` of the outer loop starts at one).

SCEV recognition At the machine code level, even very regular programs contain a large amount of "unimportant" code, such as instructions that increment loop counters or calculate addresses relative to a base pointer. Those are identified by compilers as scalar evolution expressions (SCEVs) [54, 69], as they can be expressed as functions of the canonical induction variables. Detecting SCEVs is important for two reasons: First of all, the chains of dependencies associated to their computation shall be ignored, as otherwise it greatly and unnecessarily constrains possible code transformations; Second, it allows to detect an important class of memory access patterns, the strided accesses, that are used to evaluate the profitability of some loop transformations.

Id	Polyhedron (cj,ck)	Label expression f(cj,ck)
I1 → I2	$0 \le cj \le 15, 0 \le ck \le 42$	$cj' = cj + 0ck, ck' = 0cj + ck$
I2 → I4	$0 \le cj \le 15, 0 \le ck \le 42$	$cj' = cj + 0ck, ck' = 0cj + ck$
I4 → I4	$0 \le cj \le 15, 1 \le ck \le 42$	$cj' = cj + 0ck, ck' = 0cj + ck - 1$

Table 2. Output of the folding algorithm for the dependencies stream shown in Table 1

If we detect during folding of an instruction that all the *Label* values of a polyhedron can be expressed using a SCEV, that is, if the folding succeeds in constructing an affine function to express them, the corresponding points in the folded polyhedra of IIVs as well as the associated dependencies are removed from the DDG. This happens for example for instructions I5 and I8. For instruction I5, the folding algorithm finds a SCEV with value $a(cj, ck) = 0cj + 1ck + 1$.

Over-approximations Even in programs where the hot region is affine such as in PolyBench [56], profiling the entire benchmark reveals a large amount of non-regular parts. Keeping an accurate representation of non-regular parts causes scalability issues both in the profiling part and in the polyhedral back-end of POLY-PROF. Our idea to handle such non-regular parts in a scalable fashion is through over-approximation. This over-approximation, detailed in [29], concerns labels for non-affine dependencies or non-affine SCEVs and polyhedra for dependencies and instructions with iteration domains with holes.

6 DDG Polyhedral Feedback

An essential motivation for folding DDGs into polyhedral structures is to enable the use of advanced polyhedral compilation systems, which are capable of finding a schedule that maximizes parallelism, finds tiling opportunities, etc. [11].

Polyhedral compilation of folded-DDGs Typically, a polyhedral compiler is applied to small numerical kernels made of a handful of statements [11, 24, 48, 57]. Polyhedral schedulers suffer scalability challenge for larger programs [48]: their complexity typically grows exponentially with the number of statements in the input program. Our DDG folding and over-approximation techniques allow going from programs with thousands of statements (vastly exceeding the typical program scale these schedulers can handle) to only a few hundreds.

Numerous customizations for scalability of the polyhedral compiler have been implemented, ranging from constraining the space of legal schedules to accelerate the scheduling process to approximation of the code generation process to

quickly output a decorated simplified AST describing the program structure after transformation. For example, the presence of large integer constants causes combinatorial blow up in the ILP solver used to solve the scheduling problem [52]. We implemented a parameterization of iteration domains, to replace those constants by a parameter (an unknown, but constant integer value). That is, a domain $\{[i] : 0 <= i < 1024\}$ is replaced by $[n] \longrightarrow \{[i] : 0 <= i < n \wedge n \geq 1024\}$ prior to scheduling. We control the number of parameters introduced by reusing the same parameter for a range of values, that is, if the value $x \in [1024 - s, 1024 + s]$ with $s \in \mathbb{Z}$ (we typically set $s = 20$), then we replace x by $n + (x - 1024)$.

The reader may refer to the available implementation in PoCC [55] for further details about the simplifications implemented, computation of profitability metrics is implemented in the PolyFeat module.

Final output The main visual support used for reporting aggregated feedback is the dynamic schedule-tree described in Sec. 4, along with a simplified AST that shows the code structure after the application of the suggested structured transformation. This AST embeds various metrics on instruction count, loop properties (parallelism and tilability) and the list of statements surrounded by every loop. This lets the user visualize the potential effort in manually writing the code structure corresponding to the application of the suggested transformation.

To better visualize the various program regions, metrics can weight each tree-node and be rendered with flame-graphs [26]. An example discussed in Sec. 7 is reported in Fig. 7. Here, the "surface" of a region in the flame graph is proportional to its estimated computation weight. Such flame graphs are SVG files, which can be clicked to obtain detailed information on each region/box. In the example of Fig. 7, loop/call nodes are marked with the label `loop`/`call`, respectively. Colors and gradients are used, that is, non-interesting regions can be grayed out. In Fig. 7, grayed regions are non-affine and blacklisted (initialization and extensive calls to libc) ones. Node width is used to highlight hotness of regions. In our example, functions `adjustweight` and `layerforward` take substantially more space than other regions.

Numerous additional extensions have been developed, to provide *useful* feedback to the user: various metrics about the size and structure of polyhedral regions in the DDG, before (original program) and after (transformed program) applying an optimizing schedule, the possible code structure including the precise fusion/distribution scheme, a simplified sequence of core loop transformations that should be applied to achieve the target transformation, etc. Given the extensive textual length of the feedback we provide, an example is shown only in the supplementary document.

7 Case Studies

This section aims to present case studies that illustrate both the type of feedback provided by POLY-PROF as well as the different transformations it can suggest. Current QEMU[7], on which POLY-PROF is based, cannot handle newer AVX instructions. Feedback was obtained from binaries compiled

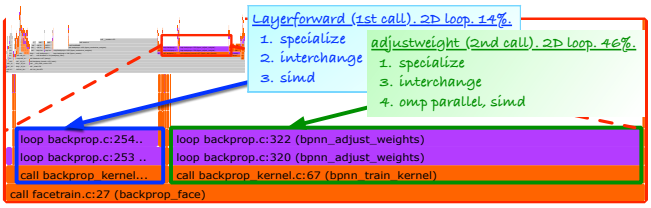


Figure 7. Annotated flame-graph for `backprop` (see Sec 7).

with GCC 8.1.1 and the flags `-g -O2 -msse3`. Speedup measurements are using Intel icc and ifort compilers (version 18.0.3, flags `-Ofast -march=native -mtune=native`) on a machine with a Xeon Ivy Bridge CPU with two 6 core CPUs (24 hyperthreads), each running at 2.1Ghz (GFlop/s are averaged over 50 runs).

Case study I *This case study illustrates a simple (structured) feedback: pinpointing that dependencies live within the first quartant [4] allows exposing data locality and fine-grain parallelism through a loop interchange.* For this study, we selected `backprop`, a supervised learning method used to train artificial neural networks extracted from the Rodinia benchmark suite [15]. In addition to what was outlined in the previous case study, POLY-PROF exploits its ability to associate an iteration vector with each memory access, and to match pointer values with scalar evolution. Since the scalar evolution expressions for a memory access describe the addresses it reads/writes, they can be used to detect strided accesses along any dimension, which in turn can be used to detect vectorization potential.

The flame-graph provided by POLY-PROF is reported in Fig. 7 where the regions of interest are: 1. the first call (of two) to `bpnn_layerfoward` with a constant value of `n2 = 16`; 2. the last call (of two) of `bpnn_adjust_weights` with a constant value of `ndelta = 16`. Both functions contain a 2D-nested-loop, called $L_{\texttt{layer}}$ and $L_{\texttt{adjust}}$ respectively, for which POLY-PROF suggests an interchange and SIMDization. For example, see $L_{\texttt{layer}}$ on Tab. 3 for which the two loop dimensions are reported to be on lines 253 (outer) and 254 (inner) of file `backprop.c`: The loop nest is fully permutable, that is, interchange is possible; only the outermost loop is parallel; and there are more stride 0/1 accesses along the outermost dimension (100%) than along the innermost (67%).

To enable the suggested transformations, one needs to specialize the two interesting function calls and array expand the scalar `sum`. The other calls to `bpnn_layerfoward` and `bpnn_adjust_weights` are left unchanged, since they (a) take up much less of the overall program runtime (b) have different arguments and do not profit from the transformation. Applying the transformation improved, in our case, `bpnn_layerfoward` from 0.5 GFlop/s to 2.8 GFlop/s and `bpnn_adjust_weights` from 0.3 GFlop/s to 5.1 GFlop/s.

Case study II *This case study illustrates an advanced feedback: providing exact dependence vector "directions" allows exposing data locality and coarse-grain parallelism through loop skewing and tiling.* For this study, we selected `GemsFDTD`, a finite difference time domain method from the SPEC CPU2006

Fat regions	% ops	interchange+SIMD	parallel	permutable	% stride 0/1	speedup
backprop_kernel.c:52 (L_{layer})	14%	backprop.c:(254,253)	(yes,no)	(yes,yes)	(100%, 50%)	5.3 x
backprop_kernel.c:57 (L_{adjust})	46%	backprop.c:(322,320)	(yes,yes)	(yes,yes)	(100%, 50%)	7.8 x

Table 3. backprop case study. Reported lines (e.g. 52, 253, ...) are from debug information. Suggested interchange is represented using a permutation of code lines. Statistics/properties per loop dimension as follows: (outer, inner).

Fat regions	op	tiling	speedup
update.F90:106	20%	update.F90:{106,107,121}	2.6 x
update.F90:240	18%	update.F90:{240,241,244}	1.9 x

Table 4. GemsFDTD case study. Reported lines are shifted debug info.

benchmark suite [32] written in Fortran90. Analyzing Fortran code is not a problem for POLY-PROF as it works at the binary level. However, the compiler we used (gfortran-8.1.1) messes up the debug information, making it necessary for the user to shift the line numbers for the provided code references by hand. This case study fully exploits POLY-PROF's ability to model (and compress in a polyhedral form) the data dependencies, instead of simply checking their existence/absence. This knowledge about the structure of dependencies allows POLY-PROF to check for tiling opportunities. First, POLY-PROF detects that four functions from the benchmark are fat (i.e. they execute a large amount of the program's total number of dynamic instructions): updateH_homo, updateE_homo, UPML_updateH, and UPML_updateE; Inside the first two of those functions are the five hottest loop nests, so we focus on them. As reported in Fig. 4, POLY-PROF annotates all five loops as fully parallel and tilable. So to obtain a speedup we tile each loop along all dimensions with a tile size of 32 and mark the outermost loop parallel with an OMP PARALLEL DO directive. We recall that tiled code can always be also coarse-grain parallelized using wavefront parallelism, as exploited by the Pluto polyhedral scheduler [11]. Tiling and parallelizing the loops increased performance in updateE_homo from 1.3 GFlop/s to 2.7 GFlop/s and updateH_homo from 1.3 GFlop/s to 3.7 GFlop/s

8 Experiments

The goal of this section is to demonstrate that POLY-PROF can be systematically applied on a full benchmark suite, and find potential for optimization. *Note that the output of* POLY-PROF *for each benchmark is extensive: flame graph, statistics on each sub-region, potential structured transformation(s), simplified annotated AST after the application of the transformation, complete AST, etc. Consequently, we only illustrate here the application of* POLY-PROF *on Rodinia by using aggregate metrics. These metrics are not meant to be used as is by the end user: instead, the user is expected to work on one benchmark at a time, and navigate the feedback we provide.*

Experiment I In addition to the case studies, we evaluate below our tool-chain using the latest revision (3.1) of the Rodinia benchmark suite [14, 15]. As POLY-PROF does not support multithreaded applications yet, each benchmark is manually modified to run in a single thread, and compiled using the same compiler and flags as the case studies (GCC 8.1.1, -g -O2 -msse3).

The dynamic analysis is based on the QEMU-plugin instrumentation interface [30]. We have extended the interface itself to support multiple interacting C/C++ plugins. Plugins primarily work at the level of the generic QEMU compiler instructions, making them CPU architecture agnostic. The choice of using QEMU for instrumenting the code is orthogonal to our contribution. There are some other candidates (e.g. Valgrind [51] or pin [45]) for implementing a dynamic binary analysis tool, each having their own advantages and disadvantages. Since we use a shadow memory to track data dependencies, dynamic analysis obviously does not come for free. As an example, the total CPU time (summing for all cores) on our server required by the first three stages of POLY-PROF to analyze the full Rodinia benchmark suite is 3h 6' (the full execution including libc was instrumented).

We have built our polyhedral feedback pass described in Sec. 6 in the PoCC compiler [55] that implements the PluTo scheduler [11], along with a new polyhedral DDG analysis and optimization in PoCC/PolyFeat [55]. Extensions for scalability have been implemented in the schedulers and code generator, and in PolyFeat.

Experiment II To show problems static approaches encounter with the Rodinia suite we also ran the static LLVM-based [40] polyhedral compiler Polly [28] over the entire suite. We used Polly version 7.0.1 and the flags -O3 -ffast-math -polly-process-unprofitable. Kernels that span multiple functions were inlined to allow Polly to see the same code region as POLY-PROF. Calls to functions from libc or the OpenMP runtime were not inlined. Where such calls are present this usually results in Polly being unable to analyze the kernel, though it can handle calls to simple functions such as exp or sqrt. Polly was unable to build a polyhedral model of the whole region of interest for any of the 19 benchmarks. It was able to model some smaller subregions, 1D or 2D loop nests, in most benchmarks, but in nearly all cases its own profitability metric decided not to optimize them. Two notable exceptions are the heartwall and lud benchmarks. In heartwall Polly was able to model a sequence of nine 2D loop nests which accounts for roughly two thirds of the code in the body of the kernel. For lud it managed to model the whole inner 3D loop nest of the kernel, but not the outermost loop. In all benchmarks the inability to model the outermost loops blocks Polly from exploiting the thread level parallelism inherent in the Rodinia benchmark suite.

Summary statistics. Table 5 presents summary statistics about the Rodinia 3.1 (CPU only) benchmarking suite, that we computed/aggregated by processing the feedback from POLY-PROF and Polly on each benchmark.

Column %Aff reports the percentage of dynamic operations that are part of a fully affine region without over-approximation. The low proportion of affine code reported

Bench.	#V	#E	%Aff	Region	%ops	%Mops	%FPops	interproc.	Reasons why Polly failed	skew	%||ops	%simdops	%reuse	%Preuse	ld-src	ld-bin	TileD	%Tilops	C	Comp.	fusion
backprop	10 M	10 M	85%	facetrain.c:25	67%	30%	76%	Y	A	N	100%	100%	50%	100%	2D	2D	2D	100%	6	4	S
bfs	2 M	1 M	21%	bfs.cpp:137	55%	66%	18%	N	BF	N	100%	100%	1%	1%	3D	3D	2D	100%	1	1	M
b+tree	23 M	24 M	49%	main.c:2345	26%	99%	0%	N	BF	N	100%	100%	44%	44%	3D	3D	3D	100%	15	4	S
cfd	251 M	372 M	98%	*3d_cpu.cpp:480	98%	42%	99%	Y	F	N	100%	61%	18%	42%	5D	4D	3D	100%	1	3	S
heartwall	4 G	8 G	1%	main.c:536	99%	38%	56%	Y	RCBF	N	100%	100%	0%	0%	7D	6D	5D	100%	1	3	S
hotspot	11 M	16 M	0%	*_openmp.cpp:318	81%	35%	89%	Y	B	Y	100%	100%	3%	3%	4D	4D	2D	100%	1	1	S
hotspot3D	210 M	256 M	99%	3D.c:261	49%	28%	81%	N	BF	N	100%	99%	11%	11%	4D	4D	3D	100%	1	1	M
kmeans	513 M	647 M	97%	*_clustering.c:160	97%	56%	98%	Y	RFA	N	100%	100%	46%	53%	4D	4D	4D	100%	1	3	S
lavaMD	879 M	1 G	0%	kernel_cpu.c:123	99%	69%	92%	N	BF	N	100%	100%	0%	0%	4D	4D	3D	100%	1	2	S
leukocyte	4 G	9 G	39%	detect_main.c:51	37%	64%	64%	Y	RCBFAP	N	100%	100%	63%	63%	4D	4D	3D	100%	11	5	S
lud	42 M	71 M	4%	lud.c:121	99%	44%	70%	Y	BF	N	99%	98%	0%	1%	5D	5D	3D	99%	3	3	S
myocyte	1 M	866 K	89%	main.c:283	99%	80%	80%	Y	CBA	N	100%	99%	47%	47%	4D	3D	1D	99%	1	3	S
nn	1 M	2 M	1%	nn_openmp.c:119	31%	42%	71%	Y	RF	N	100%	0%	0%	0%	1D	1D	1D	100%	1	1	M
nw	80 M	93 M	99%	needle.cpp:308	79%	73%	27%	Y	RF	Y	100%	100%	77%	77%	4D	4D	2D	100%	2	2	S
particlefilter	628 M	678 M	27%	*_seq.c:593	99%	5%	14%	N	CF	N	99%	100%	55%	55%	3D	3D	2D	100%	22	2	S
pathfinder	62 M	48 M	67%	pathfinder.cpp:99	31%	83%	16%	N	BP	Y	100%	0%	0%	40%	2D	2D	2D	100%	1	1	M
srad_v1	1 G	2 G	99%	main.c:241	99%	31%	93%	Y	RF	N	99%	100%	18%	18%	3D	3D	2D	100%	1	1	S
srad_v2	600 M	864 M	98%	srad.cpp:114	96%	31%	92%	Y	RF	N	100%	100%	14%	14%	3D	3D	2D	100%	1	1	S
streamcluster	779 M	1 G	97%	*_omp.cpp:1269	99%	6%	13%	Y	RCBFAP	-	-	-	-	-	6D	6D	-	-	52	-	-

Table 5. Summary statistics computed from POLY-PROF's feedback on the Rodinia benchmark suite.

for `heartwall`, `hotspot`, and `lud` is the consequence of not supporting lattices at folding time: These programs contain hand linearized nested loops whose bounds use modulo expressions and so are not recognized as fully affine. Note that, even when parts of a benchmark are not affine, we can still find affine over-approximations for those regions, and potentially find transformations for the program as a whole.

Based on the statistics provided by POLY-PROF, the biggest region for which the optimizer suggests a transformation has been selected by hand. The code reference is reported in the column Region. We considered a region to be interprocedural (column *interproc.*) if inlining was required to perform the transformation or if it contained a call to libc or the OpenMP runtime. Column %ops reports the percentage of dynamic operations of the program executed inside the region, while %Mops and %FPops reports the percentage of memory (resp. floating point) operations of the region itself. Note that the sum of %Mops and %FPops can be greater than 100% since on x86 a single instruction can both load and store to/from memory and perform an operation.

The column *Reasons why Polly failed* lists the reasons why LLVM Polly was unable to model the whole region as an affine program. They are coded as: **R.** unhandled function call; **C.** complex CFG (`break`/`return`); **B.** non-affine loop bound or non-affine conditional statements; **F.** non-affine access function (includes pointer indirection); **A.** unhandled possible pointer aliasing; **P.** base pointer not loop invariant.

The next group of metrics shows what can be achieved via semantics-preserving structured transformations. skew displays whether skewing is used in the proposed transformation, we tend to avoid skewing unless it really provides improvements in parallelism and tilability. %||ops gives the percentage of dynamic operations that can be parallelized using OpenMP parallel pragmas. If a non-inner loop dimension is detected as parallel, then all its operations are considered to be parallelizable. As a loop has at least two iterations, at least two parallel blocks can be exposed when a loop is reported parallel. Similarly, %simdops reports the percentage of operations that occur in parallel innermost loops.

The %reuse/%Preuse metrics report space locality that is available in the program: %reuse is the percentage of load/stores that are stride-0 or stride-1 in the existing innermost loops in the program, while %Preuse reports the maximal percentage of load/store operations that can be made stride-0 or stride-1 via a sequence of loop permutations.

We report the maximal loop depth of the region in the source code (*ld-src*) and in the binary code (*ld-bin*). This shows whether the compiler performed any transformation that modifies the loop depth (e.g., full loop unrolling for `cfd`). Next the maximal tiling depth (*TileD*) is reported, along with %Tilops, the percentage of operations that can be tiled.

As soon as a region can be tiled, coarse-grain (wavefront) parallelism is possible, and data reuse could be improved. POLY-PROF does not currently provide feedback on temporal locality potential, but as illustrated in the `backprop` case study %reuse allows to evaluate spatial locality improvements through tiling/interchange.

Finally, metrics C/Comp./fusion outline the complexity of the loop fusion/distribution structure that originates from the structured transformation proposed, and is an indication of the difficulty to manually implement a transformed code. Any outermost loop with more than 5% of the total region operation counts as one "component". For example, if the region is made of two consecutive loop nests executing each half of the operations, then 2 components will be reported. C reports the number of components in the binary code; Comp. the number of components after applying the proposed structured transformation, using the fusion heuristic reported in fusion (*M* for maximal loop fusion, and *S* for *smartfuse*, a somewhat balanced fusion/distribution strategy).

Note that `streamcluster` exhausted memory at the scheduling stage, and therefore no result is displayed.

9 Related Work

This section describes previous work related to profiling, performance debugging, and trace compression.

Performance feedback tools Using profiling feedback is a widely used technique both for performance debugging and for guiding optimization [64] (FDO). Tools such as Perf [18] or Intel VTune [59] gather hardware counter values and, using sampling, provide statistics (such as instruction throughput or cache miss rate) at the granularity of machine instruction. Various interfaces such as HPCToolKit [1] allow

navigating through the gathered statistics thanks to a best-effort mapping from binary to source code. Compiler based profiling tools, such as gprof [25], make it possible to provide complementary statistics (such as branch probabilities). While these allow pin-pointing important parts of the code where the programmer or compiler should focus his attention on, the metrics they report are very low level and consequently they leave the role of finding optimizing code transformations to the programmer or compiler heuristics [65]. Recent work such as in MAQAO [13, 19], AutoSCOPE [65] or MIAMI [46] combines these metrics with static analysis of binary/source code. Intel IACA [17], a purely static analysis tool, uses a precise machine model, which the vendor does not publish, to pin-point resource bottlenecks and predict performance of snippets of binary code.

Dynamic data-flow analyses have been presented to provide useful performance feedback. The detection of parallelism (such as vectorization) along canonical directions has particularly been investigated [3, 12, 21, 37, 38, 41, 67, 70, 74], as it requires only relatively localized information. Another use case is the evaluation of effective reuse [8, 10, 44, 46] with the objective of pinpointing data-locality problems.

The APOLLO [34, 47] framework uses an interesting technique that corresponds to detecting dynamic regularities (affine expressions) of memory accesses in nested loops, and use this information to perform speculative loop transformation, which differs in scope and challenges from the profiling feedback tool developed in this paper.

Polyhedral compilation Integer linear algebra is a natural formalism for representing the computation space of a loop nest. The polyhedral framework [23] leverages, among others, operators on polyhedrons (e.g. union, intersection, projection), enumeration (for code generation [6]), and parametric integer linear programming [22] (for dependence analysis [16]). Historically, it has been designed to work on restricted programming languages, and was used as a framework to perform source-to-source transformations. More recently, efforts have been made to integrate the technology in mainstream compilers (e.g. Graphite [53, 68] for GCC [27] and Polly [28] for LLVM [40]). The set of loop transformations (known as affine transformations) that the polyhedral model can perform is wide and covers most of the important ones for exposing locality and parallelism to improve performance [11]. Tools such as PoCC [55] provide a convenient interface to the numerous existing state-of-the-art scheduling heuristics and polyhedral libraries such as ISL [71].

Dynamic dependence graph Shadow memory [76] records a piece of information for each storage location used in a program. For dependency tracking this is usually the last statement or dynamic instruction that modified that location. But shadow memories are also a core component of memory error debugging tools [51, 62].

Like POLY-PROF, Redux [50] builds a data-flow graph from binary level programs. It has, however, no notion of loops or calling contexts and does not try to compress the produced graph and is consequently only able to handle very small programs. A few techniques exist to address the high overhead of monitoring data dependencies. The most common is the use of static analysis so as to remove redundant instrumentation [42], or to avoid monitoring must-dependencies [37]. Another approach that leads to over-approximation, consists of reducing the size of the shadow memory through the use of signature based addressing [43]. The most sophisticated technique is the one developed in SD^3 [39] that amounts to detecting strided memory accesses to compress the shadowing. A last technique that could be used for our purposes is the parallelization of shadowing as done in [42, 49].

Calling context tree Using calling context trees to disambiguate instructions in different calling contexts is an idea from Ammons, Ball and Larus [2]. But in the presence of recursive functions the size of their CCTs grows proportionally with the depth of the recursion, leading to an unreasonably high memory overhead. Loop-call context trees [60] simply encode the calling context of intraprocedural loops and suffer from the same size problems in recursive programs.

10 Conclusion and Future Work

In this paper we introduced POLY-PROF, a profiling-based polyhedral optimization feedback tool. POLY-PROF tackles the problem of dynamic data dependence profiling and polyhedral optimization from binary programs, while addressing the associated scalability challenges of handling traces with billions of operations, as we demonstrated. Our tool handles non-regular programs with recursive function calls. Numerous technical contributions as presented in this paper were required to enable *structured transformation feedback on binary programs*, a significant step in complexity compared to prior approaches typically limited to unstructured transformations. We implemented numerous feedback reporting schemes for the user in POLY-PROF: flame graphs, statistics on each sub-region, proposed potential structured transformation, simplified annotated AST after the application of the transformation, complete AST, etc.

While this represents an important step towards being able to provide polyhedral feedback on full-scale applications there are issues that still need to be addressed. This includes the development of under-approximation schemes in the DDG, as well as overall scalability enhancements, like using approximate (non-optimal) polyhedral scheduling strategies.

Lastly, the process of mapping our results back from the machine code back to the source code is in itself a research topic, and currently we do not provide much more than what `objdump` does. Our ongoing efforts in this direction leverage polyhedral program equivalence techniques [5, 61, 73].

Acknowledgements

This work was supported in part by the U.S. National Science Foundation awards CCF-1645514, CCF-1731612 and CCF-1750399, and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*.

[3] Ran Ao, Guangming Tan, and Mingyu Chen. 2013. ParaInsight: An Assistant for Quantitatively Analyzing Multi-granularity Parallel Region. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 698–707.

[4] Utpal K. Banerjee. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA.

[5] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 539–554.

[6] Cédric Bastoul. 2004. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron* 2 (2004), 10.

[7] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*.

[8] Erik Berg and Erik Hagersten. 2005. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 33. ACM, 169–180.

[9] Kristof Beyls and Erik H. D'Hollander. 2006. Discovery of Locality-improving Refactorings by Reuse Path Analysis. In *Proceedings of the Second International Conference on High Performance Computing and Communications (HPCC'06)*. Springer-Verlag, Berlin, Heidelberg, 220–229. https://doi.org/10.1007/11847366_23

[10] Kristof Beyls and Erik D'Hollander. 2006. Discovery of Locality-Improving Refactorings by Reuse Path Analysis. *High Performance Computing and Communications* (2006), 220–229.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *PLDI*.

[12] Khansa Butt, Abdul Qadeer, Ghulam Mustafa, and Abdul Waheed. 2012. Runtime analysis of application binaries for function level parallelism potential using QEMU. In *Open Source Systems and Technologies (ICOSST), 2012 International Conference on*. IEEE, 33–39.

[13] Andres S Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *High Performance Computing (HiPC), 2014 21st International Conference on*.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*.

[15] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*.

[16] Jean-François Collard, Denis Barthou, and Paul Feautrier. 1995. Fuzzy array dataflow analysis. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 92–101.

[17] Intel Corporation. 2009. *Intel Architecture Code Analyzer – User's Guide*.

[18] Arnaldo Carvalho de Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18.

[19] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. 2005. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, Vol. 200.

[20] Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. 2013. Spolly: speculative optimizations in the polyhedral model. *IMPACT 2013* (2013), 55.

[21] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson. 2008. Embla-data dependence profiling for parallel programming. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*. IEEE, 780–785.

[22] Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268.

[23] Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*. Springer, 1581–1592.

[24] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Intl. J. of Parallel Programming* 34, 3 (2006).

[25] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[26] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.

[27] Arthur Griffith. 2002. *GCC: the complete reference*. McGraw-Hill, Inc.

[28] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* (2012), 22:04.

[29] F. Gruber, M. Selva, D. Sampaio, C. Guillon, L.-N. Pouchet, and F. Rastello. 2019. *Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable*. Technical Report RR-9244. Inria.

[30] Christophe Guillon. 2011. Program Instrumentation with QEMU. In *Proceedings of the International QEMU User's Forum (QUF '11)*.

[31] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (July 1997).

[32] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[33] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. *ACM SIGPLAN Notices* 47, 6 (2012), 371–382.

[34] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. 2012. *VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework*.

[35] W. Kelly and W. Pugh. 1995. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*.

[36] Alain Ketterlin and Philippe Clauss. 2008. Prediction and Trace Compression of Data Access Addresses Through Nested Loop Recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*.

[37] A. Ketterlin and P. Clauss. 2012. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 437–448.

[38] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*.

[39] Minjang Kim, Nagesh B Lakshminarayana, Hyesoon Kim, and Chi-Keung Luk. 2013. SD3: An Efficient Dynamic Data-Dependence Profiling Mechanism. *IEEE Trans. Comput.* 62, 12 (2013), 2516–2530.

[40] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*.

[41] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. 2015. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*. Springer, 37–54.

[42] Zhen Li, Michael Beaumont, Ali Jannesari, and Felix Wolf. 2015. Fast data-dependence profiling by skipping repeatedly executed memory operations. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 583–596.

[43] Zhen Li, Ali Jannesari, and Felix Wolf. 2015. An efficient data-dependence profiler for sequential and parallel programs. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 484–493.

[44] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 171–180.

[45] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[46] G. Marin, J. Dongarra, and D. Terpstra. 2014. MIAMI: A framework for application performance diagnosis. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[47] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192. e4192 cpe.4192.

[48] Sanyam Mehta and Pen-Chung Yew. 2015. Improving Compiler Scalability: Optimizing Large Programs at Small Price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 143–152. https://doi.org/10.1145/2737924.2737954

[49] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE, 198–208.

[50] Nicholas Nethercote and Alan Mycroft. 2003. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 149–170.

[51] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* (June 2007).

[52] Christos H Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

[53] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*.

[54] Sebastian Pop, Albert Cohen, and Georges-André Silber. 2005. Induction Variable Analysis with Delayed Abstractions. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'05)*.

[55] Louis-Noël Pouchet. [n. d.]. the PoCC polyhedral compiler collection. ([n. d.]). http://pocc.sourceforge.net.

[56] Louis-Noël Pouchet. 2017. Polybench: The polyhedral benchmark suite (version 4.2). http://polybench.sf.net (Accessed: 2017-09-13). (2017).

[57] Benoît Pradelle, Benoît Meister, Muthu Baskaran, Athanasios Konstantinidis, Thomas Henretty, and Richard Lethin. 2016. Scalable hierarchical polyhedral compilation. In *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 432–441.

[58] Ganesan Ramalingam. 2002. On Loops, Dominators, and Dominance Frontiers. *ACM Trans. Program. Lang. Syst.* 24, 5 (Sept. 2002).

[59] James Reinders. 2005. VTune performance analyzer essentials. *Intel Press* (2005).

[60] Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. 2011. On-the-fly Detection of Precise Loop Nests Across Procedures on a Dynamic Binary Translation System. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11)*.

[61] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. 2014. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 493–508.

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[63] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. 2018. PolyJIT: Polyhedral Optimization Just in Time. *International Journal of Parallel Programming* (Aug 2018), 33.

[64] Michael D. Smith. 2000. Overcoming the Challenges to Feedback-directed Optimization (Keynote Talk). *SIGPLAN Not.* 35, 7 (Jan. 2000), 1–11.

[65] O. A. Sopeju, M. Burtscher, A. Rane, and J. Browne. 2011. AutoSCOPE: Automatic Suggestions for Code Optimizations Using PerfExpert. (July 2011).

[66] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

[67] Georgios Tournavitis and Björn Franke. 2010. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*. IEEE, 377–388.

[68] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*. ACM, Pisa, Italy.

[69] Robert A Van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*. Springer.

[70] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The Paralax infrastructure: automatic parallelization with a helping hand. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*. IEEE, 389–399.

[71] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress on Mathematical Software (ICMS '2010)*.

[72] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. (2014).

[73] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 3 (2012), 11.

[74] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael FP O'boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 2.

[75] Michael E Wolf and Monica S Lam. 1991. A data locality optimizing algorithm. In *ACM Sigplan Notices*, Vol. 26. ACM, 30–44.

[76] Qin Zhao, Derek Bruening, and Saman Amarasinghe. 2010. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 22–31.